# Scanner - A tool for collecting object oriented software metrics from C++ class definitions

Murray Wong

June 2, 1998

## 1 Introduction

This paper presents Scanner, a tool for automating the collection of object oriented software metrics from C++ class definitions. Scanner produces metrics by collecting information about classes from source code and uses this information to build up a structural representation of the class in memory. Rules on how a metric should be measured can then be applied to to the structure to produce a value for the class on that metric. In this paper we will also present the results collected from the IOS system, the current limitations of the Scanner and future enhancements that could be made to the tool.

## 2 What is Scanner

Scanner is a menu oriented tool used to aid object oriented software measurement by automating the collection of metric data from C++ source code. In addition to collecting metrics data Scanner also provides functions for querying and retrieving on metric data for a single classes from the system under analysis. Data collected by Scanner for a specified metrics for each class of a system can be exported to a file, to be used as input to data analysis packages, such as spreadsheets, for analysing and interpreting the data.

## 3 The development of Scanner

Scanner was developed with the tools g++, flex and Bison under the Unix operating system. Many of the metrics for which there is an interest to collect can be measured by directly counting a particular attribute of source code. The aim of Scanner as a metric tool is to collect information from parsing source code to produce a representation of the implementation of system in memory. Rules for collecting a particular metric can then be specified and applied to the structure to produce a value. The advantage of this approach is that once a rule on how a metric is to be collected has been specified it will be applied in a uniform fashion to all classes in a system and across systems.

Scanner is comprised of the following components:

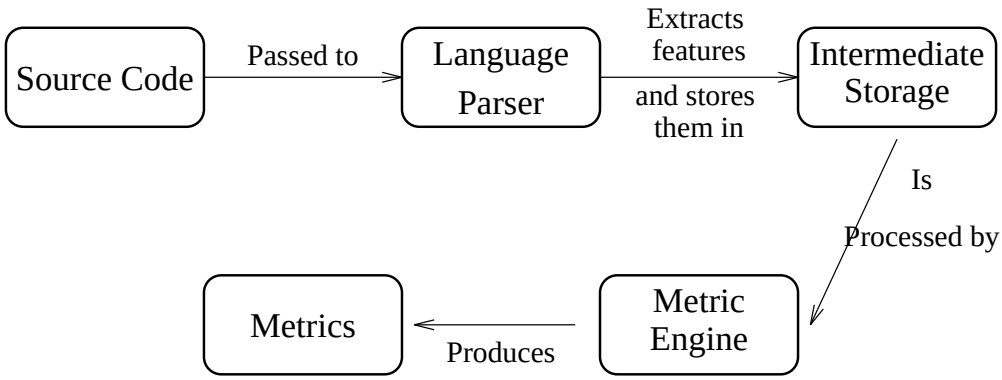


Figure 1: The components making up Scanner and the flow of processing

- A code parser - Parses source code to for class information which is stored in an intermediate storage structure to give a representation of classes in a system in memory.
- Intermediate storage structure - A list of entries where each entry in the list is a representation on a class in the system under investigation.
- A metric engine - Performs post processing of the intermediate storage structure to generates metrics for each class.
- An interface to interact with the metric engine to retrieve or export metric data for a class in the system of the system as a whole.

Figure 1 shows the processing of a system by Scanner. The information for each class is read by the parser and copied directly from the source code and stored into the intermediate storage which is actually a linked list of structures. Each entry in the linked list represents an actual class in the system being processed. Each entry in the linked list stores the following types of information about a class:

- A classes name,
- The parents of each class.
- The names of a classes methods,
- The names of a classes variables,
- The types of the variables and method arguments
- The names of any friend classes

The aim is to try to preserve the details of the class. Table 1 gives the structure of an entry in the linked list. The name of the class is used as the primary key for identifying the class. The entry's in table 1 other than the *Name* are also storage structures. The following entries are consider simple general class attributes:

| Name | Parents | Meths | MethArgs | MethReturnTypes | Vars | VarTypes | friends | Metrics |
|---|---|---|---|---|---|---|---|---|

Table 1: Structure for storing class details

- Parents - Used to link class for building up a value if the classes depth in the inheritance tree. Also used as the reference for calculating the number of children of a class.
- MethArgs - The types of the method arguments. For calculating Message passing coupling
- MethReturnTypes - The return type of each method. Used in the calculation of the violations of the law of demeter.
- VarTypes - The types of the classes instance variable. For calculating coupling through composition.
- friends - Class defined as being friends of the class. For calculating coupling from the side.

They are each stored as entries in separate linked lists of a structure with the format of table 2. Only the name of each of these attributes of a class are stored. Entries to each of these linked lists are added on the fly as the class

| Name |
|---|

Table 2: Structure for storing general class attributes

definition is being processed. The *Meths*, methods are stored in a link list of a structure with the format represented by table 3. The entries in table 3

| Name | AccessType | Overriden | Overloaded | operator |
|---|---|---|---|---|

Table 3: Structure for details about a classes method

- AccessType - Is the method private (0),protected (1) or public(2)
- Overridden - True or False
- Overloaded - True or False
- Operator - True or False

are flags that are set when a method is stored. The flag for stating a method is Overloaded is set if a method being added to the list has the same name as a method entry already in the list. The Overridden flag is set during post processing of the class after source passing is completed and a method in one of the ancestors methods linked list structure is found to have the same name. A

method is identified as an Operator at the parsing level and the Operator flag is set if the method is an operator. The *Vars* variables of a class a stored in a linked list of a structure with the format in table 4. The *AccessType* in table 4 is

| Name | AccessType |
|---|---|

Table 4: Structure for details about a classes variable

also a flag set that will be set with 0 for private, 1 for protected and 2 for public. The *Metric* part of table 1 is a structure that stores all the calculated metrics for the class. *Metric* has been defined as an enumerated structure with an entry for each of the metrics listed in section 4. An enumerated structure has been used to ease the storage and retrieval of metric values. Values are added to this structure as a post processing stage after all source files have been parsed. This is to insure all the necessary details for calculating the metrics will be available in the class linked list when the link list is processed to fill in each entry of the metric table. Many of the methods are simply calculated by counting the number of entry's in each of the linked lists.

# 4 Metrics collectible using Scanner

Scanner can be used to collect the following metrics presented in Wong98:

## 4.1 System level metrics

- Number of classes defined locally.
- Number of class hierarchies - Measured as the number of root classes in a system.

In addition Scanner can also determine for each class Chidamber and Kemerer's:

- Depth of/in the inheritance tree (DIT) - The depth the class is in the inheritance tree. If multiple inheritance is involved then the depth of the class is the maximum path from the node representing the class to the root of the tree.
- Number of children (NOC)- Number of immediate subclasses subordinate to a class in the class hierarchy.

## 4.2 Intra class metrics

The intra class metrics are measures of the attributes defined locally in a class. The attributes of a class identified by Scanner are the locally defined methods and variables. Scanner can produce the following information related to the attributes of a class.

#### 4.2.1 Measures of the classes locally defined methods

- Number of private methods (PRIM)
- Number of protected methods (PROM)
- Number public methods (PUBM)
- Number of methods that overload other (OVLM) methods in the class
- Number of methods in the class that are overrides of methods that are inherited from ancestor classes (OVRM)
- Number of methods that are overloads of operators (OPM)
- Total number of methods defined locally in a class TOTM)

#### 4.2.2 Measures of the classes locally defined variables

- Number of private variables (PRIV)
- Number of protected variables (PROV)
- Number of public variables (PUBV)
- Total number of variables defined locally in a class (TOTV)

#### 4.2.3 Inherited attributes

Scanner is also able to determine for a class the number of attributes inherited attributes i.e.

- The number of inherited methods (INHM)
- The number of inherited variables (INHV)

Scanner currently makes the assumption that the derivation or extension of classes is public in calculating the number of inherited methods and variables. The inherited attributes are taken by Scanner to be those attributes of the ancestor that are either public or protected.

### 4.3 Interclass metrics

The interclass metrics measures how many other classes a class is likely to communicate with over the duration of it's life cycle. The following forms of coupling identified by Berard can be measured from a classes definition:

- Coupling through composition/aggregation (CTC/A) - The number of locally defined class variables that are instances of other classes in the system. Multiple instances of a class are counted only once.
- Coupling through message passing (CMP) - A count of the number of arguments in a classes method prototype that are instance of other classes. Multiple instances of a class are counted only once.

- Coupling from the side (CFS) - In C++ the use of *friends* results in a class experiencing coupling from the side. Coupling from the side is the number of class friends a class has.
- Violations of the law of demeter (VLD) lieberherr89.

## 5 Output of collected data

Scanner allows the user to perform online analysis of a single class in the system or the output of data for all classes in the system in a number of formats.

In the online individual class the user has the options to display the metrics listed in section 4 for a single class by:

- Retrieval of classes by name
- Retrieval on a specific metric. The user can also specify that the value for the metric may be:
  - Equal to a certain value
  - Greater than threshold value
  - Less than a threshold value
  - Within a range specified by the entry of an upper limit and then a lower limit

The bulk generation of data for all classes in a system is provided by Scanner to produce output to a file that can then be used in analysis packages such as a spreadsheet to produce:

- Frequency distributions for each metric
- Scatter charts plotting the relation between 2 metrics
- 3D bar charts for 2 metrics in combination
- Data for producing directed graphs to represent the systems:
  - Hierarchy structures
  - Coupling through composition/aggregation structure
  - Coupling through message passing structure

## 6 Data results collected using Scanner

Scanner was applied to the IOS C++ library to give the results in table 5 for the metrics defined in section 4. The system has 18 classes and 3 class hierarchies.

| Class | DIT | NOC | CTC | CMP | CFS | VLD |
|---|---|---|---|---|---|---|
| IOS | 0 | 2 | 2 | 2 | 0 | 1 |

Table 5: Results for the interclass metrics of the IOS system

| Class | PRIM | PROM | PUBM | OVLM | OVRM | OPM | INHM | TOTM |
|---|---|---|---|---|---|---|---|---|
| IOS | 0 | 4 | 29 | 10 | 0 | 2 | 0 | 35 |

Table 6: Results for the methods of each class of the IOS system

# 7 Extending Scanner

## 7.1 Other object oriented languages

To extend Scanner to collect metrics from a language other than C++ a new parser will need to be written specifically for parsing the new target language.

## 7.2 Collecting additional class information

The various linked lists of the class structure have been defined as being *protected* types of the class. Although it is believed that the current structure of a class specified could be applied to a wide range of object oriented languages it is possible to derive from the class *classItem* a new class that may be more representative of the class structure of another target language.

## 7.3 Adding new metric definitions

Metrics collected by Scanner are calculated through the post processing of the intermediate storage structure on the completion of the parsing of the code. The main function for calculating each metric for each class in a system is called *doMetrics*, a method of class *classItem*. To insure new rules are applied to all classes in a system they should be called from within this method. To add a new metric an additional variable and entry also needs to be added to the metrics enumerated table.

# 8 Limitations of Scanner

The current implementation of Scanner was found to be unable to handle the following situations. The assumption is that the code is valid C++:

- Method definitions and normal C type functions

| Class | PRIV | PROV | PUBV | INHV | TOTV |
|---|---|---|---|---|---|
| IOS | 3 | 12 | 4 | 0 | 19 |

Table 7: Results for the methods of each class of the IOS system

- 
- Embedded classes, Structs and Enums - Although Scanner can parse these structures it does not capture the details. This may be a future enhancement.
- Templates
- Inheritance access type - Scanner does not currently make allowances for the access type of the parent class in calculating inheritance based metrics. The inherited methods of a class with a private base class are given the same treatment as the inherited methods of a class with a public base class.
- Hash defines such as "ifdef"
- Friends of a nature other than a class name for example friend methods.
- Message handling routines - Such as those defined in many of the MFC classes.
- Macros - However this problem can be overcome by first running the code through a preprocessor

# 9 Future enhancements

One of the major weaknesses currently inherent of Scanner is it's inability to parse all C++ code. The planned improvement to the tool to increase it's robustness is the implementation of another layer to perform preprocessing. The preprocessor will convert the input to a standard format before passing it to Scanners parser for gathering building up the structure.

Extending the grammar to allow for the collection of metrics from the implementation of a classes methods to gather more in depth interclass metrics is also desirable.

Although Scanner has greatly simplified the task of collecting and organizing metrics data a number of additional steps have to be taken in order to view the data in more meaningful ways such as graphically. The graphical presentations possible with the output from Scanner also limits the types of graphs that could be produced. The addition of graphical output capabilities could greatly simplify the analysis of data. The menu oriented system currently provided by Scanner is rather clumsy and at times can be very frustrating to use. The implementation of a graphical user interface could greatly simplify the use of Scanner and possibly increase the scope of an examination to more that a single metric through the use of multiple windows to display the results of different metrics simultaneously. It would also need to be implemented before any additional graphical capabilities could be considered.

Although Scanner was able to calculate values for metrics it was often found to be necessary to return to the raw source code to obtain a better interpretation of the results. Many who are unfamiliar and new to software metrics will have

a similar problem as a result of being unable to visualize the meaning of the metrics. It could be quite useful give the user the ability to retrieve and display the related piece of the source code for a class.